VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Inteligentní kamera s použitím procesoru Blackfin

Intelligent camera based on Blackfin processor

2011 David Urbášek

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Poděkování

Na tomto místě chci poděkovat vedoucímu své bakalářské práce, Ing. Michalu Krumniklovi, za cenné rady a odbornou pomoc. Dále bych rád poděkoval své rodině za trpělivost a podporu při studiu.

V Lanškrouně dne 4. 5. 2011. David Urbášek

Abstrakt

Cílem této bakalářské práce je popsat možnosti zpracování obrazových signálů na digitálním signálovém procesoru Blackfin. Práce se zabývá popisem procesoru Blackfin, jeho vstupním a výstupním rozhraním a jeho vnitřní architekturou. Součástí práce je popis propojení digitálního obrazového senzoru Kodak KAC-9618 s procesorem Blackfin. Dále je popsána implementace a optimalizace základních algoritmů pro zpracování obrazových signálů a porovnán výkon před a po optimalizaci. Posledním bodem je porovnání digitálního signálového procesoru Blackfin a běžného procesoru pro osobní počítače.

Klíčová slova

Blackfin, digitální signálové procesory, digitální zpracování obrazu, připojení obrazového senzoru k DSP, optimalizace

Abstract

The aim of this work is to describe the possibilities of image signals processing on the digital signal processor Blackfin. The work deals with the Blackfin processor description, its input and output interfaces and its internal architecture. The thesis includes the description of the interconnections of the digital image sensor Kodak KAC-9618 with the Blackfin processor. In addition, the work describes the implementation of the basic algorithms for image signal processing. The thesis compares the performance before and after optimization. The last point of the thesis is comparison of the digital signal processor Blackfin and the current personal computer.

Key words

Blackfin, digital signal processors, digital image processing, connection between the camera and DSP, optimization

Obsah

1. Úvod

V dnešní době se setkáváme stále častěji s technologiemi pro digitální zpracování signálů. Starší analogové systémy jsou, nebo již byly postupně nahrazovány moderními digitálními zařízeními téměř ve všech oblastech lidské činnosti. Nejvíce se setkáváme se systémy pro zpracování zvuku a obrazu. Příchod kompaktních disků, formátu MP3 a přenosných přehrávačů nám umožnil poslouchat oblíbenou hudbu kdykoliv a kdekoliv. Digitální fotoaparáty a digitální kamery nahradily své analogové předchůdce již téměř u všech uživatelů.

I v profesionální sféře se digitální systémy prosazují stále častěji. Zvukaři nasazují digitální mixážní pulty a procesory pro řízení reproduktorových soustav i na těch největších koncertech. Digitální systémy nalézáme v televizních studiích. V současné době vrcholí přechod z analogového televizního vysílání na vysílání digitální.

Cílem této bakalářské práce je předvést možnost propojení digitálního signálového procesoru s obrazovým senzorem a využití tohoto procesoru pro zpracování obrazu.

V úvodní kapitole budou popsány rozdíly mezi digitálním a analogovým způsobem zpracování signálů a výhody a nevýhody obou přístupů.

Následně bude popsán digitální signálový procesor Blackfin spolu s vývojovým kitem EZ-Kit Lite a vývojovým prostředím VisualDSP++. Důraz bude kladen na samotné jádro procesoru, organizaci paměti a rozhraní pro komunikaci s okolím.

Poté bude popsáno připojení obrazového senzoru Kodak KAC-9618 k procesoru, a to jak po hardwarové, tak po softwarové stránce.

V další kapitole popíši některé algoritmy, které demonstrují základní operace s obrazem. Tyto algoritmy byly nejprve napsány v jazyce C tak, aby byly přenositelné a bylo možné je přeložit a spustit i na běžném stolním počítači. Poté jsem se snažil algoritmy optimalizovat, aby co nejvíce využívaly potenciál procesoru Blackfin. Budou popsány principy optimalizace kódu v jazyce C.

Na závěr se pokusím porovnat výkon algoritmů před a po optimalizaci, a také srovnat výkon algoritmů na procesoru Blackfin a na procesoru v běžném stolním počítači.

2. Signály a jejich zpracování

Svět kolem nás můžeme popsat množstvím fyzikálních veličin, které se mohou s časem měnit. Mohou jimi být tlak vzduchu, síla magnetického pole, velikost jasu a podobně. Signálem pro účely této práce rozumíme závislost některé takové veličiny (závislá proměnná) na jiné veličině (nezávislá proměnná), typicky na čase. Signály mohou obsahovat některé, pro nás zajímavé informace. Proto máme často potřebu takové signály získávat, analyzovat, zaznamenávat, nebo dokonce upravovat.

Signály mohou měnit svoji formu. Mohou se transformovat z jedné podoby do jiné. Pro potřeby zpracování signálů nejčastěji využíváme elektronická zařízení, proto je převádíme nejčastěji do podoby elektrického napětí. V oblasti zvukových signálů bývá takový převod realizován mikrofonem, v případě obrazu je to obrazový snímač.

Systémy pro zpracování signálů dělíme do dvou kategorií. Tou první jsou systémy analogové, které pracují se signály ve spojité podobě. Jejich funkce je určena obvodovým zapojením elektronických součástek, ze kterých se skládají. Naproti tomu digitální systémy využívající pro zpracování signálů procesor, mají svou funkčnost definovanou programem uloženým uvnitř. To přináší některé podstatné výhody. Jednou z nich je jednoduchost takového zapojení. Zjednodušeně můžeme říci, že pro realizaci takového systému potřebujeme pouze procesor a vhodně napsaný program. Sériová výroba pak bývá jednodušší a levnější. Stačí pouze nahrát program do procesoru. Naproti tomu u analogových systémů musíme vždy sestavit celý obvod, který se může skládat z velkého množství elektronických součástek.

Druhou výhodou systémů využívajících procesor je jejich modifikovatelnost. Mnohdy můžeme jejich funkci změnit pouhou změnou programu, bez nutnosti zasahovat do obvodového zapojení. Výrobci těchto zařízení často poskytují na svých webových stránkách ke stažení nové verze firmwaru, tedy programového vybavení takového systému. Díky tomu je možné jeho funkčnost dodatečně opravovat, doplňovat, nebo měnit. To není v případě analogových systémů možné.

Na druhou stranu mají digitální systémy i své nevýhody. Tou hlavní je nutnost převodu signálu z analogové do digitální podoby a zpět. Analogový signál je spojitý v amplitudě i čase, nabývá tak nekonečného množství hodnot. Naproti tomu, digitální systémy pracují se signálem v čase i amplitudě nespojitým. Diskrétní signál tak může nabývat pouze omezeného počtu hodnot, který je dán vzorkovací frekvencí a bitovou hloubkou analogově-digitálního převodníku. Z tohoto důvodu nemůže signál po průchodu digitálním zřízením přesně odpovídat svému analogovému vzoru. Velkou výhodou diskrétních signálů je však jejich odolnost. Analogové signály podléhají degradaci například při přenosu na velké vzdálenosti, ukládání na záznamové médium, nebo při kopírování. To může být způsobeno útlumem na přenosovém médiu, vlivem elektromagnetického rušení, které se v okolí analogového systému vyskytuje, nebo sníženou kvalitou záznamového média. Digitální signály jsou proti těmto vlivů odolnější.

3. Digitální signálové procesory

Digitální systém pro zpracování signálů se typicky skládá ze tří částí, analogově-digitálního převodníku, procesoru a digitálně-analogového převodníku. Ne vždy jsou v systému obsaženy všechny části, například přehrávač zvukových souborů ve formátu MP3 čte data z paměti již v digitální podobě, procesor je dekóduje a následně jsou data převedena do analogové formy pomocí digitálně-analogového převodníku. Analogově-digitální převodník není v takovém případě použit. U jiných systémů sloužících k záznamu, nebo analýze signálů bývá naopak obsažen pouze analogově-digitální převodník a procesor.

Vlastní zpracování signálů má na starosti procesor, avšak ne všechny procesory se k takovým účelům hodí. Pro potřeby zpracování signálů jsou vyráběny speciální signálové procesory, které jsou navrženy a optimalizovány přímo pro tuto úlohu. Na rozdíl od jiných procesorů, určených například pro řídící aplikace, zpracovávají většinou digitální signálové procesory velké množství dat, navíc často v reálném čase. Z toho důvodu musí být digitální signálové procesory velmi výkonné a propustné.

Digitální signálové procesory jsou většinou postaveny na Harvardské architektuře. To znamená, že mají oddělenou programovou a datovou paměť i sběrnici. Tím je umožněno přistupovat do obou těchto pamětí současně, a zvýšit tak propustnost celého systému.

Další zvýšení propustnosti bývá dosahováno použitím jednotek pro přímý přístup do paměti (DMA). Díky tomu je možné přenášet velké množství dat mezi pamětí a periferiemi bez účasti samotného procesoru. Ten tak může využít svůj výpočetní výkon na provádění matematických operací a nemusí se zdržovat načítáním a ukládáním dat.

Nejvýznamnější vlastností digitálních signálových procesorů však bývá velmi výkonné jádro. To často obsahuje větší množství aritmeticko-logických jednotek, rychlé násobičky, jednotku pro bitový posun a větší množství pracovních registrů. Tyto jednotky jsou kombinovány tak, aby bylo možné v jednom instrukčním cyklu provést několik aritmetických operací, například násobení s přičítáním, což je velmi častá operace v algoritmech digitálního zpracování signálů. Dále umožňuje přítomnost většího počtu takových jednotek zpracování většího množství dat paralelně.

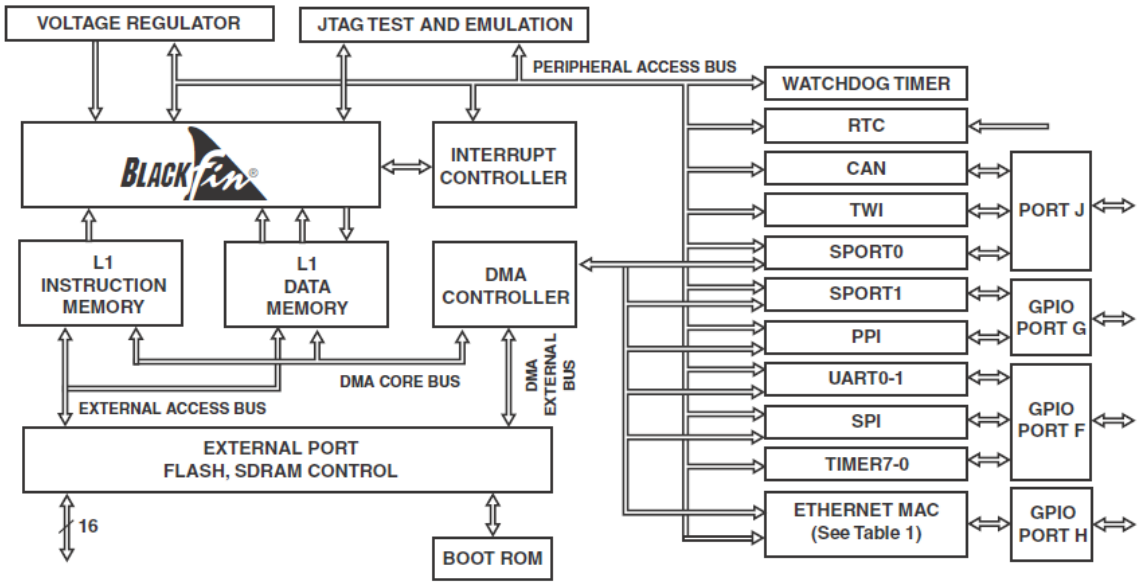
Dalšího zvýšení výkonnosti digitálního signálového procesoru může být dosaženo využitím speciálních jednotek pro generování adres, nebo jednotek umožňujících efektivní provádění cyklů.

Procesory pro zpracování signálů se dělí na dva typy. V závislosti na použité aritmetice jsou to procesory pracující s pevnou desetinou čárkou a procesory pracující s plovoucí desetinou čárkou. Procesory pracující s plovoucí desetinou čárkou bývají složitější a tím i dražší, ale nabízí větší rozsah zpracovávaných dat a větší přesnost. Naopak procesory pracující s pevnou desetinou čárkou jsou jednodušší, ale nedosahují takových výkonů při zpracování desetinných čísel.

4. Digitální signálový procesor Blackfin

Rodina digitálních signálových procesorů Blackfin je vyráběna firmou Analog Devices. Kromě řady Blackfin, nabízí tato firma ještě DPS řady Sharc a TigerSharc. Ty pracují s plovoucí desetinou čárkou a jsou určeny zejména pro náročné aplikace vyžadující větší přesnost. Na druhou stranu jsou tyto procesory dražší a mají větší spotřebu. Dále jsou v nabídce firmy signálové procesory řady SigmaDSP a ADSP-21xx. Ty pracují, podobně jako Blackfin, s fixní desetinou čárkou, mají nižší výkon, ale také nižší cenu a menší energetickou spotřebu.

Signálové procesory Blackfin jsou navrženy tak, aby kombinovaly vlastnosti digitálních signálových procesorů a klasických mikroprocesorů a umožňovaly tak nasazení v aplikacích obojího typu. Výrobce je označuje za 16/32bitové procesory, protože umí pracovat s daty o šířce 32 bitů, ale optimalizovány jsou pro práci s 16bitovými daty. Blackfin je postaven na architektuře, kterou vyvinula firma Analog Devices společně s firmou Intel a nese označení MSA (Micro Signal Architecture). Jde o architekturu s redukovanou instrukční sadou (RISC) a desetistupňovou frontou instrukcí. Délka nejčastěji používaných instrukcí je 16bitů, ale podporovány jsou i některé 32bitové instrukce. Navíc může jádro procesoru zpracovat na základě jedné instrukce větší množství dat, protože obsahuje více výpočetních jednotek. Tento princip je u procesorů označován jako SIMD (single instruction, multiple data).



4.1 - Blokové schéma procesoru Blackfin BF537

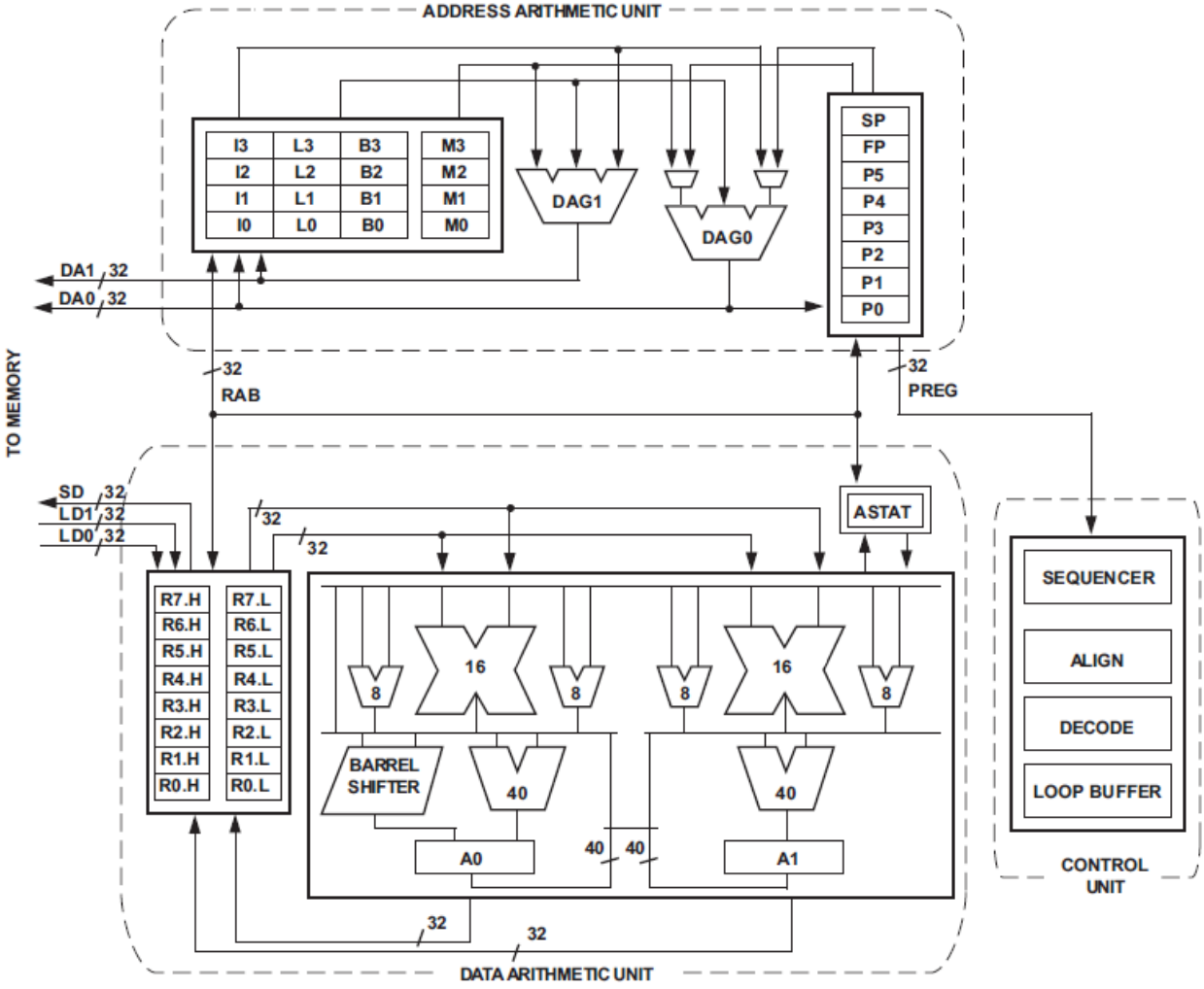
Procesory Blackfin disponují schopností dynamického řízení spotřeby. Podporují synchronní i asynchronní přerušení s možností nastavení priorit. Dále umožňují vykonávání kódu ve třech módech (User, Supervisor a Emulation). V módu User nelze přistupovat k systémovým registrům. Díky tomu je možné za použití MMU (memory managment unit) definovat a spravovat přístupová oprávnění k jednotlivým částem paměti a spouštět tak větší množství úloh bez nebezpečí jejich vzájemného ovlivňování. Tyto vlastnosti, spolu s možností připojení velké externí paměti (někdy až 512MB), umožňují spustit na tomto procesoru některé speciální operační systémy, například distribuci linuxového systémy µClinux.

V nabídce je též procesor Blackfin BF561, který obsahuje dvě shodná jádra.

4.1. Jádro procesoru

Jádro procesoru Blackfin je tvořeno dvěma 16bitovými násobičkami, dvěma 40bitvoými aritmeticko-logickými jednotkami, čtyřmi 8bitovými video ALU, dvěma 40bitovými registry pro uložení mezivýsledku, a jednou 40bitovou jednotkou pro bitový posun.

Dále je k dispozici osm 32bitových registrů pro uchování dat (R0 – R7), s nimiž jsou prováděny výpočty. Každý z osmi těchto registrů je rozdělen na dvě poloviny po 16 bitech (např. R0.H a R0.L), které mohou být použity samostatně.



4.2 - Jádro procesoru Blackfin

Každá násobička umožňuje vynásobit dvě 16bitová čísla. Výsledek pak může být za pomoci ALU přičten do 40bitového registru (A0 nebo A1), to vše v jednom jediném cyklu. Právě operace násobení s přičítáním (multiple and accumulate) je v úlohách zpracování signálů velmi častá.

Aritmeticko-logické jednotky provádí běžné aritmetické a logické operace, jako je sčítání, odečítání, logický součet a součin, negace a další. Jako operandy těchto instrukcí mohou být použity buď data z registrů R0-R7, nebo data uložená v registrech A0 a A1.

8bitové video ALU podporují vedle běžných operací také operace pro průměrování, nebo pro sčítání absolutních hodnot rozdílu operandů (subtract/absolut value/accumulate - SAA).

Procesor podporuje aritmetiku se saturací. To znamená, že umožňuje místo přetečení (resp. podtečení) nastavit jako výsledek maximální (resp. minimální) hodnotu.

Další vlastností procesoru Blackfin je přítomnost dvou jednotek pro výpočet adres (DAG0 a DAG1). Pro tyto účely jsou k dispozici 32bitové registry P0-P5 (pointer register), FP (frame pointer) a SP (stack pointer). Dále jsou zde obsaženy čtyři sady 32bitových registrů I, L, B a M (index, length, base a modify). Tím je umožněno přistupovat na dvě adresy paměti současně a zvýšit tak výkon systému. Navíc mohou výpočty adres probíhat současně s výpočty dat. Právě tyto jednotky jsou zodpovědné za přenos dat mezi vnitřními registry procesoru a pamětí.

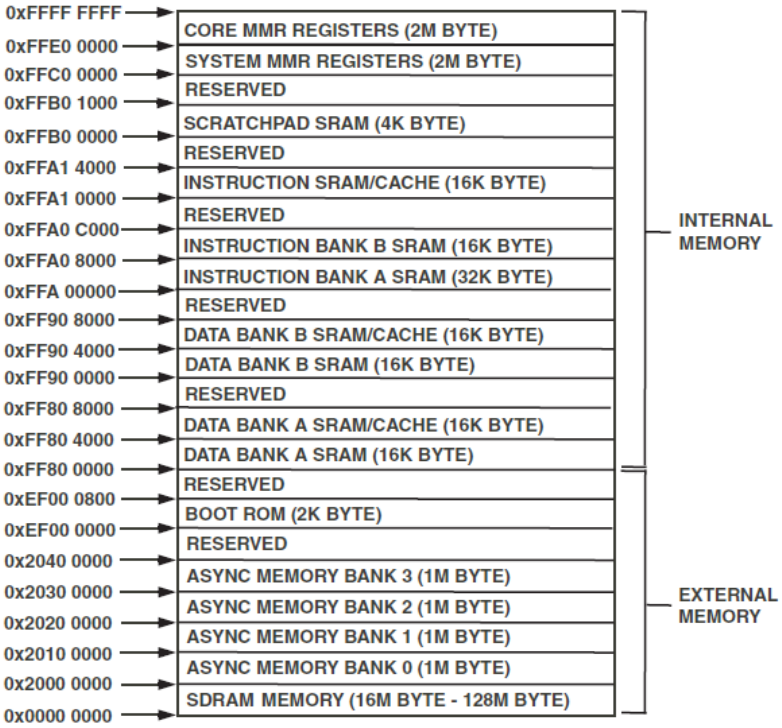
V jádru procesoru je také programový sekvencer, jednotka pro dekódování instrukcí a jednotka pro řízení průchodu cykly. Tyto jednotky se starají o řízení toku programu, volání podprogramů, podmíněné skoky (s využitím statické predikce skoku) a také umožňují efektivní průchod cyklem.

Poslední částí jádra procesoru Blackfin je registr ASTAT (arithmetic status register), který uchovává informace o výsledcích aritmetických operací, zejména pak o přetečení některých registrů.

Bližší informace o jádru procesoru Blackfin jsou k dispozici (1)

4.2. Organizace paměti

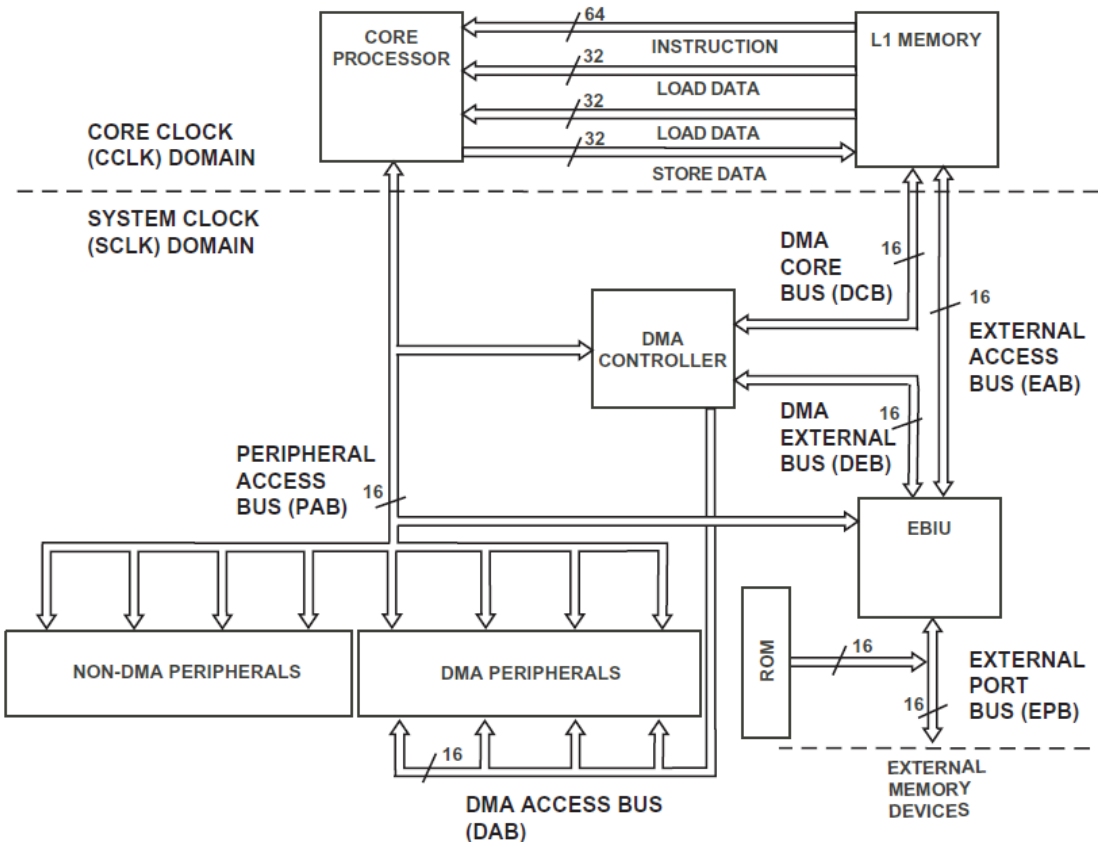
Procesor Blackfin definuje jednotný adresní prostor o rozsahu 4GB (šířka adresy je 32 bitů). V tomto prostoru jsou mapovány všechny typy pamětí (interní i externí), dále pak vstupní a výstupní periferie a také registry MMR (memory-mapped registers).



4.3 - Organizace paměti procesoru Blackfin BF537

Paměť procesoru Blackfin je organizována hierarchicky. Nejblíže jádru procesoru je paměť typu L1, která pracuje na stejné taktovací frekvenci, jako jádro a nabízí tak nejvyšší výkon. Tato paměť je

složena ze tří částí, programové, datové a paměti typu Scratchpad RAM. Programová i datová část L1 paměti jsou navíc rozdělené na dvě další části, z nichž jedna pracuje vždy v režimu SRAM, zatímco u druhé lze volit mezi režimem SRAM a cache. Architektura mezi jádrem a pamětí typu L1 je tak velmi podobná Harvardské architektuře. Kromě oddělení programové a datové paměti jsou odděleny i jejich sběrnice, takže lze přistupovat do obou pamětí současně. Velikosti jednotlivých bloků paměti se v závislosti na typu procesoru liší.



4.4 - Sběrnice procesoru Blackfin BF537

Některé typy procesorů Blackfin navíc obsahují i paměť typu L2. Ta pracuje, podobně jako paměť L1, na stejné taktovací frekvenci jako jádro procesoru, ale přístup k ní již není tak rychlý a trvá obvykle více instrukčních cyklů. Na rozdíl od paměti L1 je zde již využita Von-Neumannova architektura, to znamená, že je tato paměť společná pro data i instrukce.

Procesor Blackfin podporuje rovněž připojení externí paměti, které je realizováno přes 16bitové rozhraní EBIU (External Bus Interface Unit) a umožňuje připojit synchronní i asynchronní paměť. Pro připojení asynchronní paměti jsou k dispozici čtyři paměťové banky o velikosti 1MB, přes které lze přistupovat k pamětem typu EPROM, ROM, SRAM, nebo flash. Kontrolér pro synchronní paměť typu DRAM (SDRAM) je kompatibilní se standardem PC133 a umožňuje připojit paměť o velikosti od 16MB do 128MB do jedné banky (některé typy procesorů Blackfin mají až čtyři banky).

Pro zvýšení propustnosti a výkonnosti systému nabízí Blackfin taktéž několik kanálů pro přímý přístup do paměti (DMA), které umožňují přenášet data mezi interní pamětí procesoru, externími paměťmi a periferiemi podporujícími DMA (např. SPORT, PPI, SPI, UART apod.). Díky tomu je možné přenášet data bez zatížení jádra procesoru.

4.3. Periferie

Procesory rodiny Blackfin nabízí mnoho typů vstupně výstupních rozhraní pro komunikaci s okolím. Implementovaná rozhraní se v jednotlivých typech procesorů liší. Následuje výčet těch nejzajímavějších.

- GPIO
- PPI (Parallel Peripheral Interface)
- SPORT
- SPI (Serial Peripheral Interface)
- UART (Universal Asynchronous Receiver/Transmitter)
- Ethernet
- CAN (Controller Area Network)
- TWI (Two Wire Interface)
- USB

4.3.1. GPIO

Vývody procesoru Blackfin jsou organizovány jako porty F, G, H a J (platí pro procesor Blackfin BF537). Všechny tyto vývody (s výjimkou portu J) mohou být ovládány individuálně programátorem. Každý z nich může být nastaven jako vstupní nebo výstupní a lze přistupovat ke každému zvlášť. Tento režim je nazýván GPIO (General-Purpose Input/Output). Alternativně jsou tyto vývody použity pro některé z dalších rozhraní.

Přiřazení jednotlivých rozhraní k portům (a jednotlivým pinům) lze nalézt (2)

4.3.2. Rozhraní PPI

Procesor Blackfin disponuje paralelním rozhraním PPI. Jde o polo-duplexní obousměrné rozhraní o šířce 16 bitů. Vedle 16 datových pinů jsou zde tři piny pro synchronizaci a jeden pro přenášení hodinového signálu. Rozhraní PPI podporuje standardy ITU-R 601/656.

Toto rozhraní je vhodné například pro připojení A/D a D/A převodníků, video dekodérů a enkodérů, LCD displejů, nebo obrazových senzorů.

4.3.3. Rozhraní SPORT

Rozhraní SPORT je synchronní sériové obousměrné rozhraní. Pro vstup dat jsou k dispozici dva nezávislé datové piny, jeden synchronizační pin a jeden pin pro přenos hodinového signálu. Stejné je to i pro výstup. Vstupy i výstupy mají k dispozici buffer.

Toto rozhraní je vhodné zejména pro připojení A/D a D/A převodníků pomocí standardu I^2S. Taktéž jsou podporovány standardy G.711, H.100, H.110, MVIP-90 a HMVIP.

4.3.4. Rozhraní SPI

SPI je taktéž synchronní sériové obousměrné rozhraní. Pro přenos dat jsou k dispozici dva piny (pro každý směr jeden), další pin slouží pro přenos hodinového signálu. Na rozdíl od rozhraní SPORT se v tomto případě jedná o sběrnici, na kterou může být připojeno větší množství zařízení. Řídící zařízení poté určuje, se kterým podřízeným zařízením bude komunikovat. Pro tento účel musí mít podřízené zařízení ještě jeden pin, sloužící pro jeho výběr (SS pin – Select Slave). Řídící zařízení musí mít minimálně tolik pinů SS, kolik je k němu připojených podřízených zařízení. Výběr zařízení je realizován nastavením příslušného SS pinu na log. 0.

4.3.5. Rozhraní UART

Procesor Blackfin podporuje taktéž standardní rozhraní UART. Jde o asynchronní oboustranné sériové rozhraní, které podporuje přenos 5 až 8 bitů. Dále může být přenesen paritní bit (sudá nebo lichá parita). Následuje jeden nebo dva stop bity.

Toto rozhraní rovněž podporuje standard IrDA. Pro vysílání na sběrnici EIA-232 a EIA-485 je potřebné použít externí obvod pro přizpůsobení vysílaného signálu elektrickým parametrům těchto linek.

4.3.6. Rozhraní TWI

Rozhraní TWI plně podporuje standard I^2C. Jedná se o sběrnici s možností připojení většího počtu zařízení, z nichž minimálně jedno pracuje v režimu *Master* a minimálně jedno v režimu *Slave*. Komunikace je synchronní, obousměrná a polo-duplexní. Rozhraní TWI se sestává ze dvou pinů, jedním pro přenos hodinového signálu a jedním pro přenos dat.

4.4. Vývojové prostředky pro práci s procesorem Blackfin

4.4.1. Vývojový kit EZ-Kit Lite

Firma Analog Devices nabízí několik typů vývojových přípravků EZ-Kit Lite, které slouží pro podporu vývoje aplikací na platformě Blackfin. Popsán bude kit s procesorem BF-537.

Tento kit obsahuje dvě externí paměti. Synchronní SDRAM o velikosti 64MB a asynchronní paměť Flash o velikosti 4MB.

K dispozici je tlačítko Reset, které resetuje všechny integrované obvody na desce (s výjimkou USB rozhraní). Dále jsou zde čtyři tlačítka připojená na rozhraní GPIO, která mohou být použita k libovolnému účelu. Na desce je také osazeno šest LED, opět připojených na GPIO, LED signalizující přítomnost napájecího napětí, LED signalizující Reset a LED signalizující USB komunikaci.

Vývojový kit obsahuje konektory pro audio vstup a výstup (stereo jack). Ty jsou připojeny k rozhraní SPORT0 přes A/D, resp. D/A převodník.

Dále je k dispozici řada dalších konektorů (napájení, síťové rozhraní, USB konektor a další) a přepínačů (sloužících zejména pro nastavení). (3)

4.4.2. VisualDSP++

Pro vývoj softwaru nabízí firma Analog Devices integrované vývojové prostředí VisualDSP++, jehož 90denní zkušební verzi je možné stáhnout z webových stránek firmy.

Samozřejmostí tohoto IDE je podpora psaní kódu (zvýrazňování syntaxe, automatické odsazování apod.), správa projektů, správa sestavení a další běžné funkce.

Důležitou součástí je pak simulátor různých typů procesorů Blackfin. Díky tomu je možné psát, testovat a ladit programy bez použití fyzického procesoru. Simulátor umožňuje zobrazovat aktuální stavy většiny registrů a obsah paměti. Navíc poskytuje nástroje pro interpretování různých typů dat. Lze tak zobrazovat obrázky a vykreslovat grafy na základě dat uložených v paměti. Dále je možné simulovat vstupní a výstupní datové přenosy a asynchronní přerušení. (4)

Právě tento simulátor byl použit pro implementaci a optimalizaci grafických algoritmů, které budou popsány dále.

5. Připojení obrazového snímače k procesoru

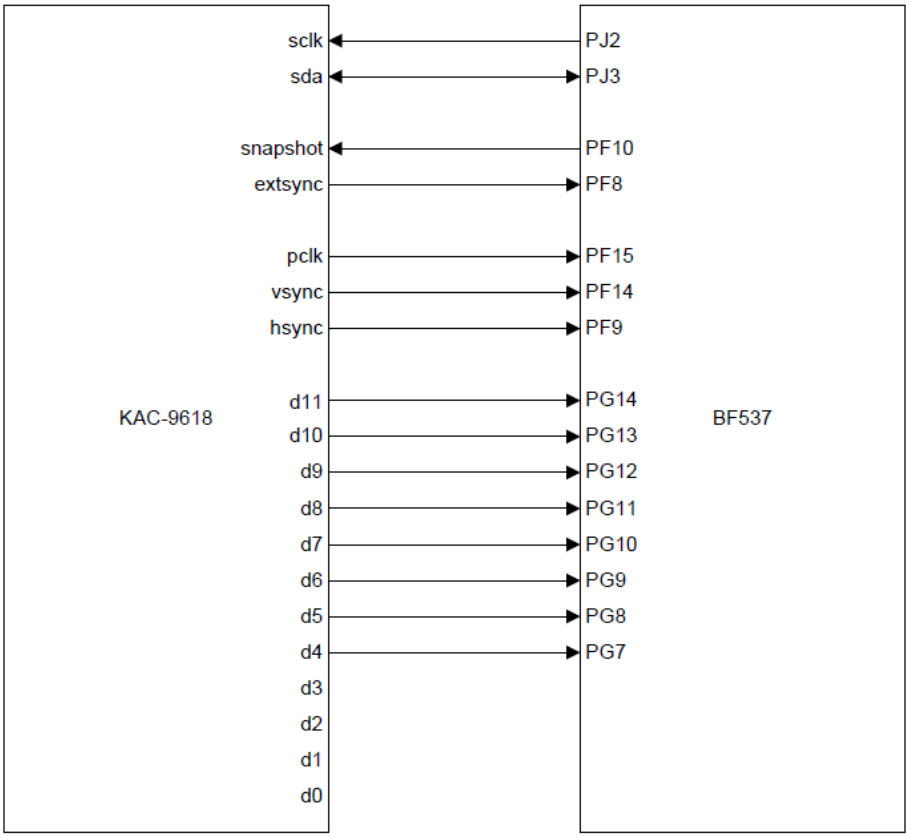
V minulé kapitole byla popsána některá vstupně-výstupní rozhraní, která je možné použít pro připojení obrazového snímače k procesoru Blackfin. Nyní bude popsán použitý obrazový senzor a jeden z možných způsobů jeho připojení k procesoru.

5.1. Obrazový senzor Kodak KAC-9618

Použitý obrazový senzor KAC-9618 je černobílý senzor typu CMOS, který obsahuje integrovaný 12bitový A/D převodník. Jeho velikost je 1/3" a obsahuje 648 x 488 aktivních bodů. Velikost jednoho bodu je 7,5µm. Snímač může pracovat v režimech *video* a *snapshot*. V režimu video funguje jako videokamera, která poskytuje sekvenci snímků rychlostí 30 snímků za vteřinu. V režimu *snapshot* funguje jako fotoaparát, jednotlivé snímky jsou poskytovány na požádání. Snímky mohou být přenášeny prokládaně i neprokládaně. Lze pořizovat výřezy, řídit zisk snímače a mnoho dalšího.

Podrobný popis funkcí tohoto senzoru lze nalézt (5)

Důležité jsou možnosti komunikace snímače s okolím. Pro nastavení parametrů je k dispozici rozhraní I^2C. Obrazová data jsou přenášena paralelně, synchronně. Vedle dvanácti datových pinů a jednoho pinu pro přenos hodinového signálu jsou k dispozici také piny pro vertikální a horizontální synchronizaci.



5.1 - Propojení obrazového senzoru a procesoru Blackfin BF535

5.2. Připojení řídící sběrnice

Obrazový snímač disponuje rozhraním I^2C, pomocí něhož lze snímač konfigurovat. Na straně procesoru Blackfin se jako vhodný způsob připojení nabízí rozhraní TWI, které rovněž podporuje standard I^2C a je vyvedeno na port J.

Při komunikaci s obrazovým snímačem bude procesor pracovat v režimu *master* a snímač v režimu *slave*. Procesor Blackfin obsahuje řadu registrů pro nastavení a řízení komunikace pomocí rozhraní TWI. Pro programátory v jazyce C jsou k dispozici hlavičkové soubory, ve kterých jsou definovány ukazatele na tyto registry a další pomocné konstanty, které lze pro komunikaci použít.

Pro komunikaci pomocí sběrnice I^2C v režimu *master* je potřeba nejprve nastavit následující registry:

- TWI_CONTROL
  Povolení komunikace pomocí bitu TWI_ENA
  Nastavení parametru PRESCALE (poměr mezi systémovou taktovací frekvencí a interní frekvencí kontroléru)
- TWI_CLKDIV
  Nastavení frekvence sběrnice
- TWI_MASTER_ADDR
  Nastavení cílové adresy, přičemž bit určující režim čtení/zápis se nenastavuje (kontrolér si ho doplní sám podle nastaveného směru přenosu v registru TWI_MASTER_CTL)

Postup pro příjem dat je následující:

- Nastavením registru TWI_MASTER_CTL se přenos zahájí (je třeba nastavit minimálně množství přijímaných dat, směr přenosu a zapnout režim *master*)
- Ukončení přenosu jednoho (resp. dvou) bytu lze detekovat pomocí bitu MCOMP v registru TWI_INT_STAT.
- Po dokončení přenosu jednoho (resp. dvou) bytu je třeba tyto data vyzvednout z registru TWI_XMT_DATA8 (resp. TWI_XMT_DATA16)

```
char I2C_receive(char addr){

  *pTWI_FIFO_CTL |= XMTFLUSH; // Vycisteni bufferu

  // Vycisteni od chybovych zprav
  *pTWI_MASTER_STAT = BUFWRERR | BUFRDERR | DNAK | ANAK | LOSTARB;
  ssync();
  *pTWI_FIFO_CTL = 0; // Nastaveni manualne
  *pTWI_CONTROL = TWI_ENA | PRESCALE120M; // Zapne TWI a nastavi PRESCAKE
  *pTWI_CLKDIV = CLKLOW(100) | CLKHI(100); // Nastavi rychlost prenosu
  *pTWI_MASTER_ADDR = addr; // Cilova adresa

  // Zahajeni prijmu jedne zpravy (8 bitu)
  *pTWI_MASTER_CTL = (1 << 6) | MEN | MDIR;
  // Ceka na dokonceni prenosu
  while ((*pTWI_INT_STAT & MCOMP) == 0)
    ssync();

  *pTWI_INT_STAT = XMTSERV | MCOMP;  // Nastaveni TWI pro dalsi prenos

  return  *pTWI_RCV_DATA8;  // Vrati prectena data
}
```

Postup pro vysílání je pak následující:

- Do registru TWI_XMT_DATA8 (případně TWI_XMT_DATA16) se uloží vysílaná data
- Nastavením registru TWI_MASTER_CTL se přenos zahájí (je třeba nastavit minimálně množství vysílaných dat, směr přenosu a zapnout režim *master*)
- V případě, že je vysílaných dat více, je potřeba zajistit jejich předávání do bufferu. Stav bufferu lze zjistit z registru TWI_FIFO_STAT.
- Ukončení přenosu jednoho (resp. dvou) bytu lze detekovat pomocí bitu MCOMP v registru TWI_INT_STAT.

```
void I2C_transmit(unsigned short *data, char addr, int dataSize){

  int j;

  // Vycisteni bufferu
  *pTWI_FIFO_CTL |= XMTFLUSH;
  // Vycisteni od chybovych zprav
  *pTWI_MASTER_STAT = BUFWRERR | BUFRDERR | DNAK | ANAK | LOSTARB;

  ssync();

  // Nastaveni manualne
  *pTWI_FIFO_CTL = 0;
  // Zapne TWI a nastavi PRESCAKE
  *pTWI_CONTROL = TWI_ENA | PRESCALE120M;
  // Nastavi rychlost prenosu
  *pTWI_CLKDIV = CLKLOW(100) | CLKHI(100);
  // Cilova adresa
  *pTWI_MASTER_ADDR = addr;
  // Nastaveni prenasenych dat
  *pTWI_XMT_DATA8 = *data++;

  // Zahajeni prenosu jednoho Bytu
  *pTWI_MASTER_CTL = (dataSize << 6) | MEN;

  // Pro vsechny Byty
  for (j = 0; j < (dataSize-1); j++){
    // Dokud je buffer plny, prenasi se
    while (*pTWI_FIFO_STAT == XMTSTAT)
      ssync();
    // Az je prenesen 1B a uvolni se tim misto v bufferu doplnime ho
    *pTWI_XMT_DATA8 = *data++;
    ssync();
  }

  // Ceka na dokonceni prenosu
  while ((*pTWI_INT_STAT & MCOMP) == 0)
    ssync();

  // Nastaveni TWI pro dalsi prenos
  *pTWI_INT_STAT = XMTSERV | MCOMP;
}
```

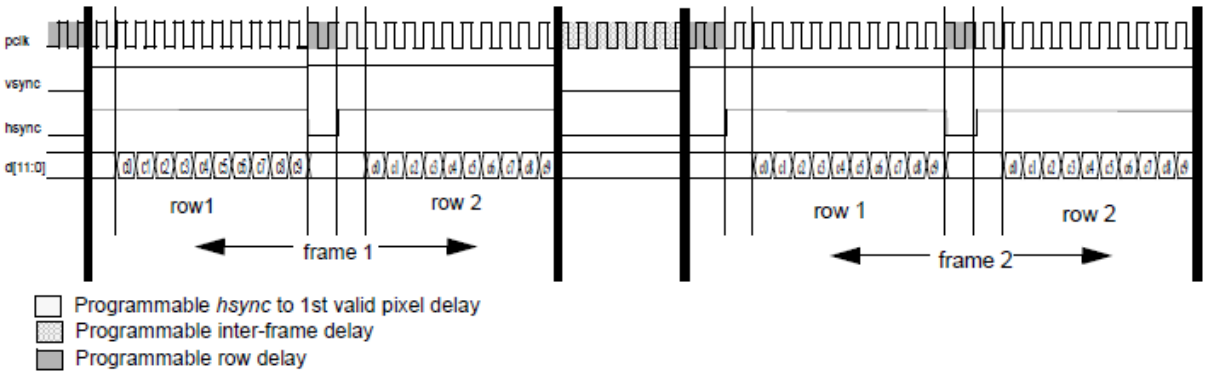
5.3. Připojení datové sběrnice

Obrazový snímač poskytuje snímky v digitální podobě prostřednictvím synchronního paralelního rozhraní. Šířka sběrnice je 12 bitů. Protože však grafické algoritmy popsané v další kapitole pracují s 8bitovými obrázky, bude pro přenos využito pouze horních 8 bitů. Dále jsou přenášeny signály vertikální a horizontální synchronizace a samozřejmě hodinový impuls. Hodinové impulsy budou generovány obrazovým snímačem. Na straně procesoru bude použito rozhraní GPIO, to znamená, že veškeré řízení přenosu bude prováděno softwarově. Podobně jako v případě rozhraní TWI budou použity konstanty a makra definované v příslušných hlavičkových souborech.

Před zahájením přenosu dat je potřeba nejdříve nastavit příslušné vstupy a výstupy. Řídící a synchronizační signály budou připojeny na port F, zatímco datové signály budou připojeny na port G.

```
*pPORTF_FER    = 0x0000;      //Cely port F jako GPIO
*pPORTFIO_DIR = 0x0400;      //0000 0100 0000 0000(0-in, 1-out)

                              //PF15 - PCLK        IN
                              //PF14 - VSYNC       IN
                              //PF10 - SNAPSHOT    OUT
                              //PF 9 - HSYNC       IN
                              //PF 8 - EXTSYNC     IN

*pPORTFIO_INEN  = 0xC300;    //1100 0011 0000 0000
                              //(0-input disabled, 1-input enabled)

*pPORTFIO_POLAR = 0x0000;    //polarita

*pPORTG_FER      = 0x0000;   // Cely port G jako GPIO
*pPORTGIO_DIR    = 0x0000;   //0000 0000 0000 0000
*pPORTGIO_INEN   = 0xFFFF;   //poveleny všechny vstupy
*pPORTGIO_POLAR = 0x0000;    // polarita
```

Obrazový snímač Kodak KAC-9618 nabízí několik možností chování synchronizačních signálů v průběhu přenosu obrázku. Pro přenos v našem případě bude použito základní nastavení. Při něm je nejprve nastaven bit VSYNC na úroveň log. 1 a v tomto stavu zůstává po celou dobu přenosu obrázku. Dále je nastaven na úroveň log. 1 bit HSYNC, který zůstává v tomto stavu po dobu přenosu jednoho řádku. Nakonec je na úroveň log. 1 nastaven bit PCLK, který tím indikuje přítomnost dat na výstupních pinech.



5.2 - Protokol pro přenos obrázku

Kód pro přenos jednoho obrázku je následující:

```
int row = 0, column = 0;
//Ceka na novy obrazek
while(((*pPORTFIO) & 0x4000)==0);
while(((*pPORTFIO) & 0x4000)!=0){
  //Ceka na novy radek
  while((*pPORTFIO & 0x0200) == 0);
  while((*pPORTFIO & 0x0200) != 0) {
    //Ceka na novy pisel
    while((*pPORTFIO & 0x8000)==0);
    while((*pPORTFIO & 0x8000)!=0);
    //Ulozi pixel (>>7 provede posun na PG7, coz je nejnizsi bit obrazku)
    image[row][column] = (unsigned char)((*pPORTGIO & 0x7FF8) >> 7);
    //incrementace ukazatele
    column++;
  }
  row++;
  column = 0;
}
```

6. Algoritmy pro zpracování obrazu

Digitální zpracování obrazu se používá pro řešení různých typů úloh. Prvním typem může být zlepšení kvality obrazu, který byl špatně pořízen (například kvůli špatným světelným podmínkám, nekvalitnímu snímači, apod.). Do této kategorie patří například algoritmy pro úpravu jasu, vyhlazení, nebo naopak zostření obrazu.

Pokročilejší metody zpracování obrazu bývají využity v případech, kdy má být obraz interpretován počítačem, například při výstupní kontrole strojírenských výrobků, při automatickém porovnávání otisků prstů, nebo třeba při rozpoznávání poznávacích značek automobilů. K tomuto účelu slouží například algoritmy pro detekci hran, prahování, nebo korelace. (6)

V této kapitole bude předvedeno několik základních algoritmů pro zpracování obrazu, napsaných v jazyce C za použití simulátoru ve vývojovém prostředí VisualDSP++.

Uvedené zdrojové kódy ukazují pouze nejdůležitější části algoritmů, nikoli celé funkce. Nejsou v nich uvedeny deklarace proměnných ani podmínky hlídající meze polí a podobně. Kompletní zdrojové kódy jsou k dispozici jako elektronická příloha této práce.

6.1. Digitální reprezentace obrazu

Z programátorského hlediska chápeme obraz jako dvourozměrné pole obrazových bodů (pixelů), kde jeden rozměr představuje řádky a druhý rozměr sloupce (případně jeden rozměr představuje osu x a druhý osu y). Samotný obrazový bod potom uchovává hodnotu úrovně jasu (v případě černobílého obrazu), případně hodnoty úrovní jasů jednotlivých barevných složek (v případě barevného obrazu v modelu RGB, nebo CMY).

Algoritmy popsané v této kapitole pracují s černobílým obrazem, kde úroveň jasu je reprezentována 8bitovým číslem. Obraz bude reprezentován strukturou, která zahrnuje dvourozměrné pole obrazových bodů a rozměry obrazu.

```
typedef struct {
  unsigned char data_array[][];
  unsigned short width;
  unsigned short height;
} IMAGE;
```

6.2. Kopírování obrázku

První implementovaný algoritmus slouží pro kopírování obrázků. Ten bude využit v některých následujících algoritmech. Je velmi jednoduchý a je zde uveden pouze proto, že bude předmětem optimalizace popsané v další kapitole.

```
//Pro všechny radky a sloupce
for (row = 0; row < height; row++)
  for (column = 0; column < width; column++)
    //Zkopiruj pixel
    dst->data_array[row][column] = src->data_array[row][column];

dst->width = src->width;
dst->height = src->height;
```

6.3. Negativ obrázku

Algoritmus pro jasovou inverzi (negativ) je jedním z nejjednodušších algoritmů zpracování obrazu. V případě černobílého obrázku obsahuje každý pixel hodnotu úrovně jasu. Úkolem algoritmu je vypočítat opačnou hodnotu. To znamená, že místo hodnoty 0 bude nová hodnota 255 (maximum pro 8 bitů) a naopak. Obecně je inverzní hodnota určena vztahem: $h' = 255 - h$

```
for(row = 0; row < image->height; row++)
  for(column = 0; column < image->width; column++)
    image->data_array[row][column] = 255 - image->data_array[row][column];
```

6.1 - Porovnání originálního obrázku a negativu

6.4. Gamma korekce

Gamma korekce je operace pro úpravu jasu, při které se křivka úrovně jasu mění z lineárního průběhu na průběh přibližně odpovídající vztahu: $h' = h^{1/k}$, kde k je koeficient gamma korekce. Algoritmus potřebuje hodnoty jasu v rozsahu <0,1>.

```
for(row = 0; row < image->height; row++)
  for(column = 0; column < image->width; column++){
    pixel_value =
      pow((double)image->data_array[row][column]/255.0, koeficient) * 255;

    if(255 < pixel_value)
      image->data_array[row][column] =  255;
    else
      image->data_array[row][column] =  (unsigned char)pixel_value;
  }
```

6.2 - Porovnání originálního obrázku a obrázku po gamma korekci s koeficientem 0,5

6.5. Výpočet histogramu

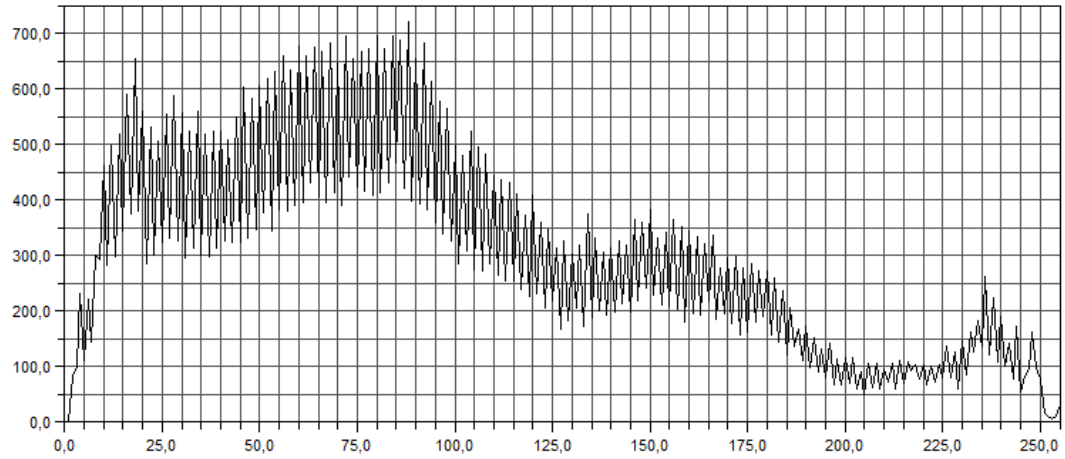
Histogram představuje četnost výskytu jednotlivých jasových úrovní v obrázku. Je důležitý jak pro získání informací o obrázku (například u digitálních fotoaparátů pomůže odhalit špatnou expozici), tak pro následné úpravy obrázku.

Algoritmus pro výpočet histogramu spočívá v průchodu obrázkem a spočítání množství výskytu jednotlivých jasových úrovní.

```
for(row = 0; row < image->height; row++)
  for(column = 0; column < image->width; column++)
    histogram[image->data_array[row][column]]++;
```

Alternativou k takovému histogramu je normalizovaný histogram, ve kterém je četnost výskytu jednotlivých úrovní jasu normována do intervalu <0,1>. Toho je dosaženo vydělením každé hodnoty počtem bodů obrázku.

```
for(i = 0; i < 256; i++)
  histogram[i] /= (image->width * image->height);
```



6.3 - Histogram originálního obrázku

6.6. Vyrovnání histogramu

Algoritmus vyrovnání histogramu způsobí, že všechny jasové úrovně budou v obrázku zastoupeny se stejnou četností. Tím je obvykle zvýšen kontrast obrázku a jednotlivé struktury v něm jsou více zřetelné (zvláště, pokud je originální obrázek příliš tmavý, nebo příliš světlý).

Metoda pro vyrovnání histogramu nejprve vypočítá normalizovaný histogram původního obrázku. S jeho pomocí jsou následně vypočítány nové hodnoty úrovní jasu podle vztahu: $h_i = h_{i-1} + n_i$, kde i je původní hodnota úrovně jasu, h_i je nová hodnota, nahrazující původní hodnotu i, h_{i-1} je nová hodnota předešlé jasové úrovně a n_i je počet výskytů hodnoty i zjištěný z histogramu.

```
for(i = 0; i < 256; i++){
  if(i == 0)
    new_values_n[i] = histogram_n[i];
  else
    new_values_n[i] = new_values_n[i-1] + histogram_n[i];
}
```

Nové hodnoty jsou teď normovány do intervalu <0,1> a je potřeba je převést do intervalu <0,255>, proto se musí vynásobit hodnotou 255.

```
for(i = 0; i < 256; i++)
  new_values = new_values_n[i] * 255.0;
```

Nyní již pouze stačí nahradit stávající hodnoty novými.

```
for(row = 0; row < image->height; row++)
  for(column = 0; column < image->width; column++)
    image->data_array[row][column] =
      new_values[image->data_array[row][column]];
```

6.4 - Porovnání originálního obrázku a obrázku po vyrovnání histogramu

6.7. Vyhlazení průměrováním

Vyhlazení průměrováním je první algoritmus, který pro výpočet nové úrovně jasu daného bodu využívá úrovně jasů bodů ve svém okolí. K tomuto účelu je využit princip konvoluce.

Pro výpočet konvoluce je nutné definovat konvoluční jádro (masku). Jde o dvourozměrné pole obsahující hodnoty (koeficienty). Pro výpočet nové hodnoty obrazového bodu se maska přiloží na dané místo obrazu (střed masky se kryje se zkoumaným bodem), poté jsou mezi sebou vynásobeny pod sebou ležící hodnoty a výsledky sečteny. Výsledkem je nová hodnota úrovně jasu zkoumaného bodu.

```
//Zaklad metody apply_kernel

int posun_radku = kernel_height / 2;      //posun v obrazku
int posun_sloupce = kernel_width / 2;     //posun v obrazku

//pro vsechny radky masky
for(i = 0; i < kernel_height; i++)
  //pro vsechny sloupce masky
  for(j = 0; j < kernel_width; j++)
    suma += image->data_array[radek - posun_radku + i]
      [sloupec - posun_sloupce + j] * kernel[i][j];
```

Tímto způsobem se konvoluční maska aplikuje na všechny body obrázku a výsledná hodnota se uloží do nového obrázku. To je nutné, jinak by se další hodnoty nespočítaly správně, protože by počítali s hodnotami již změněnými.

```
for(i = 0; i < image->height; i++)
  for(j = 0; j < image->width; j++)
    new_image->data_array[i][j] = apply_kernel(kernel,
      kernel_width, kernel_height, original_image, i, j);
```

Konvoluce nám umožňuje provádět velké množství grafických operací pouhou změnou konvolučního jádra. Jádro pro vyhlazení průměrováním může vypadat takto:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Na obrázku lze porovnat výsledek aplikace takové konvoluční masky s původním obrázkem.

6.5 - Porovnání originálního a vyhlazeného obrázku

6.8. Detekce hran pomocí operátoru Laplacian of Gaussian

Konvoluci lze využít také k detekci hran. Na hranách obvykle dochází k prudkým změnám úrovně jasu. Pro nalezení hrany tak lze využít derivaci obrazu. Nalezení vertikálních hran je možné použitím následující masky:

-1	0	1
-1	0	1
-1	0	1

Tato maska provede první derivaci obrazu v horizontálním směru, čímž zvýrazní hrany na tento směr kolmé.

Následující maska je kombinací Gaussova operátoru, který provádí vyhlazení obrazu a Laplaceova operátoru pro druhou derivaci:

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

Výsledek aplikace operátoru Laplacian of Gaussian lze vidět na následující ukázce.

6.6 - Porovnání originálního obrázku a obrázku po detekci hran

6.9. Prahování s distribucí chyby

Prahování převádí obrázek o více úrovních jasu na obrázek, který má pouze dvě úrovně (typicky černá a bílá). Rozhodující je v tomto případě úroveň zvoleného prahu. Všechny body mající hodnotu jasu nižší než zvolený práh jsou nastaveny na minimální hodnotu, naopak všem bodům, které mají hodnotu jasu vyšší, je nastavena maximální hodnota.

Prahování s distribucí chyby je zvláštní případ prahování. Používá se v případech, kdy je možné obraz reprezentovat pouze pomocí dvou úrovní, ale zároveň je nutné, aby obraz působil co nejvěrněji.

Principem tohoto algoritmu je zapamatovat si chybu (hodnotu, o kterou byl daný pixel zesvětlen, resp. ztmaven) a tuto chybu následně přesunout do okolních obrazových bodů (odečíst, resp. přičíst). Přitom se chyba nepřesouvá do již zpracovaných bodů.

```
for(row = 0; row < image->height; row++)
  for(column = 0; column < image->width; column++){
    //Hodnota je vetsi, nebo rovna
    if(image->data_array[row][column] >= treshold){
      //Vycisleni chyby
      error = image->data_array[row][column] - 255;
      //Nastaveni maximalni hodnoty
      image->data_array[row][column] = 255;
    }
    //Hodnota je mensi
    else{
      //Vycisleni chyby
      error = image->data_array[row][column];
      //Nastaveni minimalni hodnoty
      image->data_array[row][column] = 0;
    }
    //Distribuce chyby do vedlejsich pixelu
    image->data_array[row][column + 1] += 0.375 * error;
    image->data_array[row + 1][column + 1] += 0.125 * error;
    image->data_array[row + 1][column] += 0.375 * error;
    image->data_array[row + 1][column - 1] += 0.125 * error;
  }
```

Výsledek operace prahování s distribucí chyby lze vidět na následujícím obrázku.

6.7 - Porovnání originálního obrázku a obrázku po prahování s distribucí chyby

7. Optimalizace

Algoritmy pro zpracování obrazu bývají často výkonově velmi náročné. Výpočet hodnoty jednoho obrazového bodu ještě nemusí být zásadně problematický, těchto bodů je však v obrázku hodně, a proto je důležité se optimalizaci algoritmů věnovat. To je zejména nutné u systémů pracujících s videosignálem v reálném čase, kdy je potřeba zajistit zpracování aktuálního snímku dříve, než dorazí snímek následující. V této kapitole budou popsány základní techniky optimalizace algoritmů napsaných v jazyce C.

Aby bylo možné napsaný kód optimalizovat, je nutné pochopit architekturu systému, na kterém bude tento kód vykonáván. V tomto případě se jedná o procesor Blackfin, jehož architektura byla popsána dříve. Znalost vlastností cílového procesoru dává určitou představu o tom, jak rychlého kódu lze dosáhnout (například kolik cyklů procesoru je nutných pro výpočet jednoho pixelu) a jakým způsobem. Pomocí prostředí VisualDSP++ je možné sledovat vykonávání kódu v jazyce assembler a zjistit, jestli tento kód využívá možností procesoru skutečně efektivně.

7.1. Využití optimalizátoru

Prvním krokem pro dosažení rychlejšího kódu je využití optimalizátoru. Algoritmy napsané v jazyce C jsou standardně překládány bez optimalizace. Překladač firmy Analog Devices však disponuje velmi silným optimalizátorem, navrženým speciálně pro procesory této firmy. V prostředí VisualDSP++ lze optimalizátor zapnout v menu *Project Options – Compile – General*. Je zde možnost zvolit poměr mezi rychlostí a velikostí programového kódu, nastavit chování vložených funkcí a zapnout mezi procedurální analýzu (IPA - interprocedural optimization). Pokud je IPA vypnuta, optimalizátor se vždy soustředí pouze na jednu konkrétní proceduru a není schopen vnímat její kód (a data, se kterými kód pracuje) v souvislostech celého programu. Například není schopen zjistit, že proměnná použitá uvnitř metody je definována jako konstanta. Zapnutím IPA může optimalizátor využít informace o datech k dosažením rychlejšího kódu.

Za předpokladu, že ani pomocí optimalizátoru nedosahuje kód požadované rychlosti, je zapotřebí do něj zasáhnout ručně. Pro nalezení nejslabších míst v kódu lze využít *profiler* vývojového prostředí VisualDSP++. Tento nástroj měří čas a počet procesorových cyklů strávených na jednotlivých řádcích, čímž ukazuje, kde je potřeba s optimalizací začít. Bližší informace o možnosti měření procesorových cyklů lze nalézt (7)

V první řadě je potřeba kód v jazyce C přepsat do takové podoby, aby byl pro optimalizátor co nejvíce čitelný. Obecné zásady pro psaní takového kódu budou uvedeny v této kapitole. Pokud ani po těchto úpravách není optimalizátor schopen dosáhnout dostatečně efektivního kódu, je možné napsat kritické části algoritmů v assembleru. (8)

7.2. Obecné zásady pro psaní kódu v jazyce C

7.2.1. Aritmetika v plovoucí desetinné čárce

Procesor Blackfin nedisponuje jednotkami pro práci s čísly v plovoucí desetinné čárce. Všechny takové operace musí být emulovány softwarově, a proto jsou velmi časově náročné. Například operace dělení dvou desetinných čísel trvá přibližně 250 procesorových cyklů. Z tohoto důvodu je nutné se operacím v plovoucí desetinné čárce úplně vyhnout, nebo je alespoň zredukovat na minimum.

Někdy lze s výhodou použít datové typy *fract16* a *fract32* a funkce pro práci s nimi. Jejich podpora je zabudována přímo v překladači firmy Analog Devices. Tyto typy reprezentují desetinná čísla v pevné řádové čárce v rozsahu <0,1> a procesor s nimi pracuje stejně, jako s celými čísly, takže jejich výpočty jsou velmi rychlé.

7.2.2. Využití konstant

Kompilátor umí nahradit proměnnou, která má v průběhu celého programu pouze jednu hodnotu, konstantou, a zvýšit tak efektivitu kódu. Aby to bylo možné, je potřeba konstantní proměnnou inicializovat již při její definici. Tím je zajištěno, že bude mít skutečně pouze jednu hodnotu po celou dobu běhu programu.

```
//definice a soucasna inicializace konstantni promenne
const int promenna = 10;
```

7.2.3. Umístění dat v paměti

Pro efektivní načítání a ukládání dat je důležité, aby byla data v paměti správně zarovnána. 16bitová data by měla být vždy na sudých pozicích. 32bitová data na pozicích dělitelných čtyřmi. O zarovnání některých dat (například globálních polí) se stará optimalizátor, jindy je však nutné zarovnání vynutit konstrukcí *#pragma align*.

Druhou důležitou věcí je umístění dat, ke kterým se přistupuje v jeden okamžik, v odlišných paměťových sekcích. Díky tomu je pak možné přistupovat k těmto datům současně.

7.2.4. Použití vložených funkcí

Pokud je v kódu často volána nějaká krátká funkce, může být výhodnější tuto funkci deklarovat jako vloženou, pomocí klíčového slova *inline*. Při překladu je volání takové funkce nahrazeno jejím kódem, takže při běhu programu již k volání nedochází. Nedochází tak ani k výkonnostním ztrátám vzniklým voláním funkce (předání parametrů, uchování návratové hodnoty). Navíc, volání funkce uvnitř cyklů znemožňuje použít hardwarovou podporu procesoru pro rychlý průchod cykly.

7.2.5. Optimalizace podmínek

Podmínky v jazyce C jsou v assembleru reprezentovány instrukcí podmíněného skoku. V případě, že je podmínka pro skok splněna, dojde k přesunu toku programu na jinou adresu. Pokud podmínka splněná není, pokračuje procesor ve zpracovávání následujících instrukcí. Skok na jinou adresu však zapříčiní výpadek fronty instrukcí. Ta se musí začít plnit znovu a tím dochází k výkonnostním ztrátám. Jednou z možností, jak instrukce skoku omezit, je sdělit překladači, jaká větev podmínky se bude provádět častěji (je více pravděpodobná). Tato větev programu pak bude následovat přímo po instrukci skoku a skok se provede pouze pro větev, která nastává méně často.

K tomuto účelu jsou k dispozici vestavěné funkce *expected_true* a *expected_false*, díky kterým lze překladači sdělit, jaký výsledek podmínky je pravděpodobnější.

7.2.6. Optimalizace cyklů

Při zpracování signálů tráví algoritmy obvykle nejvíce času uvnitř cyklů. Proto je jejich optimalizace velmi důležitá. Optimalizátor se snaží najít nejefektivnější variantu zpracování cyklu, ale aby tak mohl učinit, je potřeba napsat kód v co nejjednodušší formě a vyhnout se snahám o ruční optimalizaci. V opačném případě je pro optimalizátor velmi složité takový kód pochopit. V jazyce C jsou k dispozici direktivy, které mohou napovědět optimalizátoru, jak cyklus zpracovat.

Direktivou *#pragma loop_count* lze sdělit optimalizátoru, kolikrát bude cyklus proveden. Je možné zadat minimální a maximální počet průchodů cyklem, případně že počet průchodů bude vždy dělitelný určenou hodnotou.

Direktiva *#pragma vector_for* říká optimalizátoru, že cyklus je možné vektorizovat. Díky tomu je lze v průběhu jednoho průchodu cyklem zpracovat větší množství dat paralelně.

Direktiva *#pragma different_banks* informuje o tom, že zpracovávaná data jsou v rozdílných paměťových sekcích a lze je tak načítat souběžně.

K dispozici jsou i další direktivy, které je možné najít (9)

Tělo cyklu by mělo být vždy co možná nejmenší. Je-li kód uvnitř cyklu příliš komplexní, je pro optimalizátor těžší takový kód zpracovat. Jsou-li cykly vnořené, měl by vnitřní cyklus probíhat vícekrát než vnější.

Procesor Blackfin disponuje podporou pro efektivní průchod cyklem, kdy takový průchod nevyžaduje žádnou přídavnou režii. Takto je možné zpracovávat až dva cykly (jeden je uvnitř druhého). Aby bylo možné tuto vlastnost použít, nesmí uvnitř cyklů docházet k volání funkcí a taktéž se nesmí uvnitř cyklů nacházet podmíněné výrazy.

Podrobné informace o optimalizaci kódu v jazyce C lze nalézt (9)

7.3. Optimalizace grafických algoritmů

7.3.1. Kopírování obrázku

Algoritmus pro kopírování obrázků ve své základní podobě kopíruje jeden obrazový bod v jednom průchodu cyklem. Optimalizátor dokáže průchod polem vektorizovat, takže po zapnutí optimalizace zkopíruje algoritmus dva body při jednom průchodu cyklem.

Protože algoritmus pracuje se dvěma poli, jejich umístěním do rozdílných paměťových bank je umožněn přístup do obou současně. Druhou úpravou je deklarace rozměrů polí jako konstantních proměnných, díky čemuž lze lépe optimalizovat průchod cykly. Poslední úpravou je přidání direktiv pro optimalizaci vnitřního cyklu.

```
//deklarace obrazku
IMAGE section("sdram0_bank2") image;
IMAGE section("sdram0_bank3") new_image;
```

```
//jádro funkce na kopirovani (image = src, new_image = dst)
const int height = src->height;
const int width = src->width;

for (row = 0; row < height; row++)
  #pragma different_banks
  #pragma vector_for
  for (column = 0; column < width; column++)
    dst->data_array[row][column] = src->data_array[row][column];

dst->width = src->width;
dst->height = src->height;
```

Výsledkem provedených úprav je zkopírování čtyř obrazových bodů při jednom průchodu cyklem.

7.3.2. Negativ obrázku

Algoritmus pro vypočítání inverze obrázku lze opět vektorizovat, podobně jako v předchozím případě. Tentokrát je však vektorizace provedena ručně. Velikosti polí jsou opět deklarovány konstantně.

```
const int full = 0xffffffff;
const int height = image->height;
const int width = image->width;

for (row = 0; row < height; row++){
  //pole je vektorizovano rucne - zpracovava se 32b zaroven
  for (column = 0; column < width; column += 4) {
    *((int *) (&image->data_array[row][column])) =
      full - *((int *) (&image->data_array[row][column]));
  }
}
```

Protože obrazové body leží v poli za sebou, je možné místo jednoho bodu (8 bitů), načíst rovnou čtyři body (32 bitů) a provést inverzi všech čtyř bodů v jednom kroku. Výsledkem jsou čtyři zpracované

body za jeden průchod cyklem, oproti jednomu bodu v případě původního algoritmu. Průchod cyklem navíc vyžaduje méně procesorových cyklů.

7.3.3. Gamma korekce

Hlavním problémem algoritmu gamma korekce jsou výpočty v plovoucí desetinné čárce. Pro výpočet každého obrazového bodu je nutné provést jedno dělení, jedno násobení a dokonce jedno umocnění. Zpracování jednoho obrazového bodu tak trvá přibližně 7700 procesorových cyklů, což je opravdu hodně.

Protože má však typický obrázek podstatně více obrazových bodů, než kolik je možných úrovní jasu (ukázkový obrázek má 76800 bodů, zatímco úrovní jasu je pouze 256), je výhodnější nejprve vypočítat nové úrovně jasů a poté pouze vyměnit stávající hodnoty za nové. Druhou změnou je drobná ekvivalentní úprava původního vzorce pro novou hodnotu, která odstraňuje jedno násobení. Hodnoty velikosti polí jsou opět uloženy v konstantních proměnných.

```
const float inv_koeficient = 1 / koeficient;
const float multiple = pow((double) 255, (double) 1 - inv_koeficient);

unsigned char new_values[255];

//Predpocitame si tabulku novych hodnot
int i, j, pom;
for (i = 0; i < 256; i++) {
  pom = (int)(pow(i, inv_koeficient) * multiple);
    if(pom > 255)
      new_values[i] = 255;
    else
      new_values[i] = pom;
}

const int height = image->height;
const int width = image->width;
int row, column;

//Vymena starych hodnot za nove
for (row = 0; row < height; row++)
  for (column = 0; column < width; column++) {
    image->data_array[row][column] =
      new_values[image->data_array[row][column]];
  }
```

Výsledkem těchto úprav je přibližně třistanásobné zrychlení algoritmu na obrázku velikosti 240 x 320 bodů. Je zřejmé, že čím větší by obrázek byl, tím by bylo zrychlení výraznější.

7.3.4. Výpočet histogramu

U algoritmu pro výpočet histogramu jsou opět velikosti polí uchovány v konstantních proměnných. Dále je použita direktiva pro vektorizování vnitřního cyklu. Žádné zásadní zrychlení však u tohoto algoritmu nelze očekávat.

Naopak při výpočtu normalizovaného histogramu je situace jiná. Původní algoritmus používal aritmetiku v plovoucí desetinné čárce (součet desetinných čísel), a to není z hlediska výkonu vhodné.

Protože hodnoty normalizovaného histogramu jsou v intervalu <0,1>, lze pro výpočet a reprezentaci normalizovaného histogramu s výhodou využít datový typ *fract* a funkce pro práci s ním.

```
fract32 inv_image_volume =
  float_to_fr32(1.0 / (float)(image->width * image->height));

int i, row, column;

for (i = 0; i < 256; i++)
  hist_n[i] = 0;

for(row = 0; row < image->height; row++)
  #pragma vector_for
  for (column = 0; column < image->width; column++)
    hist_n[image->data_array[row][column]] =
      add_fr1x32(hist_n[image->data_array[row][column]], inv_image_volume);
```

V důsledku těchto úprav je výpočet normalizovaného histogramu prakticky stejně rychlý, jako výpočet klasického histogramu. Počet procesorových cyklů nutných pro zpracování jednoho bodu se snížil přibližně třicetkrát (z původních zhruba 150 na nynějších 5).

7.3.5. Vyrovnání histogramu

Při optimalizaci algoritmu pro vyrovnání histogramu je nutné navázat na změny provedené v předešlém algoritmu. Programový kód musí být přepsán tak, aby místo aritmetiky v plovoucí desetinné čárce využíval funkce pro práci s datovým typem *fract*. Zbytek algoritmu je již stejný, pouze jsou opět doplněny konstantní proměnné pro velikosti polí.

```
//Vypocita nove hodnoty urovni v intervalu <0,1>
for (i = 1; i < 256; i++)
  new_values_n[i] = add_fr1x32(new_values_n[i - 1], hist_n[i]);

//Prevede nove hodnoty do intervalu <0, 255>
for (i = 0; i < 256; i++)
  new_values[i] =
    (unsigned char)(fr32_to_float(new_values_n[i]) * (float)255);

const int height = image->height;
const int width = image->width;

//Nahradi stavajici urovne novyma
for (row = 0; row < height; row++)
  for (column = 0; column < width; column++)
    image->data_array[row][column] =
      new_values[image->data_array[row][column]];
```

Zrychlení dosažené těmito úpravami však není nijak zásadní, protože se týká pouze cyklu pro výpočet nových úrovní jasů. Mnohem důležitější je zrychlení dosažené optimalizací předešlého algoritmu, který je zde volán.

7.3.6. Vyhlazení průměrováním

Původní algoritmus pro vyhlazení průměrováním využíval, společně s algoritmem pro detekci hran, univerzální metodu *apply_kernel*. Tato metoda musela umět pracovat s konvoluční maskou obsahující jak celočíselné, tak reálné koeficienty. Proto pracovala v aritmetice s plovoucí desetinou čárkou. Z hlediska výkonu může být výhodnější použít pro vyhlazení průměrováním speciální metodu, která bude pracovat s celočíselnými hodnotami.

```
//Metoda pro vypocet prumerne hodnoty z okolnich bodu
inline unsigned char get_avarage_value(const IMAGE *image,
  const int width, const int height,
  const int radek, const int sloupec) {

  const int posun_radku = height / 2; //posun v obrazku
  const int posun_sloupce = width / 2; //posun v obrazku
  const int pocet_bodu = width * height;

  int i, j, suma = 0;

  //pro vsechny radky kernelu
  for (i = 0; i < height; i++)
    //pro vsechny sloupce kernelu
    for (j = 0; j < width; j++)
      suma += image->data_array[radek - posun_radku + i]
        [sloupec - posun_sloupce + j];

    return (unsigned char) (suma / pocet_bodu);
}
```

Tato metoda nepřebírá jako argument konvoluční masku, ale pouze počet okolních bodů, ze kterých má udělat průměr. Není proto tak univerzální, ale za to je mnohem rychlejší. Navíc je deklarována jako vložená, takže nedochází ke zdržení při jejím volání.

7.3.7. Detekce hran pomocí operátoru Laplacian of Gaussian

Algoritmus pro detekci hran využívá původní funkci *apply_kernel*, avšak přepsanou do celočíselné aritmetiky.

```
unsigned char apply_kernel(char *kernel[],
  unsigned short kernel_width, unsigned short kernel_height,
  IMAGE *image, unsigned short radek, unsigned short sloupec) {

  int i, j; //odpovidaji radku a sloupci v kernelu

  int posun_radku = kernel_height / 2; //posun v obrazku
  int posun_sloupce = kernel_width / 2; //posun v obrazku
  int suma = 0;

  //pro vsechny radky kernelu
  for (i = 0; i < kernel_height; i++)
    //pro vsechny sloupce kernelu
    for (j = 0; j < kernel_width; j++)
      suma += image->data_array[radek - posun_radku + i]
        [sloupec - posun_sloupce + j] * kernel[i][j];

  if (expected_false(suma > 255))
    return 255;
  else if (suma < 0)
    return 0;
  else
    return (unsigned char) suma;
}
```

Díky tomu, že funkce již neprovádí žádné operace v plovoucí desetinné čárce, je mnohem rychlejší. Volání funkce *apply_kernel* je, až na doplnění konstant pro velikosti polí, stejné.

7.3.8. Prahování s distribucí chyby

U algoritmu prahování s distribucí chyby dochází opět k výpočtům v plovoucí desetinné čárce, kterým se je potřeba vyhnout. V původním kódu docházelo k opakovanému počítání jedné osminy a třech osmin z vyčíslené chyby. Je vhodnější si tyto hodnoty spočítat pouze jednou. Hlavně je však možné vyhnout se výpočtům v plovoucí desetinné čárce.

```
if (image->data_array[row][column] >= treshold) {
  error = image->data_array[row][column] - 255;
  image->data_array[row][column] = 255;
} else {
  error = image->data_array[row][column];
  image->data_array[row][column] = 0;
}

//Spocitani hodnot chyby pro distribuci
error_right_down = (3 * error) / 8;
error_diagonal = error / 8;

//Distribuce chyby
image->data_array[row][column + 1] += error_right_down;
image->data_array[row + 1][column + 1] += error_diagonal;
image->data_array[row + 1][column] += error_right_down;
image->data_array[row + 1][column - 1] += error_diagonal;
```

Provedené úpravy jsou jen drobné, avšak snižují počet nutných cyklů na zpracování jednoho obrazového bodu z původních přibližně 1350 na nynějších přibližně 35, a to je velmi podstatný rozdíl.

8. Porovnání výkonu

8.1. Porovnání výkonu algoritmů před a po optimalizaci

Pro měření výkonu grafických algoritmů před a po ruční optimalizaci byl použit *profiler* vývojového prostředí VisualDSP++. Ten umožňuje změřit počet cyklů procesoru nutných k vykonání kódu. Při měření algoritmů byl v obou případech zapnut optimalizátor, mezi procedurální analýza a byla použita vyrovnávací paměť.

Algoritmus	Před optimalizací	Po optimalizaci
Kopírování	178 592	142 026
Negativ	556 091	85 842
Gamma korekce	606 708 800	2 010 384
Histogram	482 878	415 779
Normalizovaný histogram	11 087 460	416 218
Vyrovnání histogramu	12 454 924	862 531
Vyhlazení průměrováním	199 961 000	2 214 209
Detekce hran	408 752 000	9 110 752
Prahování s distribucí chyby	103 616 640	2 560 408

8.1 - Počet procesorových cyklů před a po optimalizaci

Výsledky ukazují velký rozdíl ve výkonnosti, především u algoritmů pracujících s desetinnými čísly. Je vidět zejména zásadní zrychlení algoritmu gamma korekce, který sice využívá aritmetiku v plovoucí desetinné čárce, avšak četnost těchto výpočtů byla při optimalizaci výrazně snížena. Taktéž použití specializovaných funkcí v případě algoritmů pro vyhlazení průměrováním a detekci hran (oproti původně použité jedné univerzální), přispělo k zásadnímu zrychlení. V případě algoritmu pro výpočet negativu lze vidět zásadní zrychlení díky současnému výpočtu čtyř obrazových bodů.

Popsané grafické algoritmy by jistě bylo možné optimalizovat ještě více, například přepsáním kritických částí do assembleru. Předvedené způsoby optimalizace však ukazují, že i relativně jednoduchými zásahy lze dosáhnout někdy i mnohonásobného zrychlení.

8.1. Porovnání výkonu procesoru Blackfin a běžného procesoru pro osobní počítače

V předchozí kapitole byla popsána optimalizace grafických algoritmů. Původní kód byl napsán obecně a je možné ho spustit na libovolné platformě. S použitím tohoto kódu je tak možné porovnat výkonnost procesoru Blackfin s běžným procesorem.

Pro toto porovnání byl použit procesor Intel Mobile Celeron Dual-Core T1400. Jedná se o dvoujádrový procesor určený pro přenosné počítače. Každé jádro procesoru pracuje na frekvenci 1,733GHz, efektivní rychlost vnější sběrnice je 533MHz. Šířka zpracovávaných dat je 64 bitů. Procesor disponuje 64 kB instrukční L1 cache, 64 kB datové L1 cache a dále 512 kB sdílené L2 cache. Taktéž implementuje podporu instrukcí MMX, SSE, SSE2, SSE3 a EM64T. (10)

Procesory lze porovnávat podle mnoha kritérií. Jedním z nich je počet procesorových cyklů, nutných k vykonání programového kódu. Na straně procesoru Blackfin lze tuto hodnotu získat velmi snadno použitím *profileru* vývojového prostředí VisualDSP++. Zjistit počet cyklů provedených procesorem běžného počítače je obtížnější, protože tento procesor vykonává, kromě zkoumaného kódu, ještě mnoho jiných činností (běh operačního systému a dalších aplikací). Pro zjištění počtu cyklů byl nakonec použit *profiler* vývojového prostředí Microsoft VisualStudio 2010 Ultimate. Výsledky získané tímto nástrojem nejsou zcela přesné, pro porovnání je však přesnost dostatečná.

Algoritmus	Blackfin	Intel T1400
Kopírování	178 592	984 563
Negativ	556 091	960 538
Gamma korekce	606 708 800	45 859 800
Histogram	482 878	1 314 900
Normalizovaný histogram	11 087 460	1 401 770
Vyrovnání histogramu	12 454 924	2 502 150
Vyhlazení průměrováním	199 961 000	38 938 400
Detekce hran	408 752 000	88 192 600
Prahování s distribucí chyby	103 616 640	11 992 300

8.2 - Počet procesorových cyklů na procesoru Blackfin a Intel T1400

Z naměřených údajů vyplývá, že procesor Blackfin dosahuje vynikající výkonnosti v algoritmech, které pracují pouze s celými čísly (kopírování, negativ, výpočet histogramu). Jakmile jsou však v algoritmech přítomny výpočty v plovoucí desetinné čárce, procesor Blackfin za běžným procesorem zaostává. Důvodem je nutnost tyto výpočty emulovat, zatímco procesory pro osobní počítače umí v plovoucí desetinné čárce pracovat přirozeně.

Na druhé straně je ale potřeba zhodnotit i další kritéria. Jedním z nich je energetická spotřeba, která může být za jistých okolností důležitější než výpočetní výkon, a která je u běžného procesoru mnohonásobně vyšší. V neposlední řadě bude také systém postavený na platformě procesoru Blackfin výrazně levnější, než řešení s běžným procesorem pro osobní počítače.

9. Závěr

Cílem této bakalářské práce bylo popsat možnosti zpracování obrazových signálů na digitálním signálovém procesoru Blackfin. Nejprve byl popsán tento procesor, jeho vnitřní architektura a jeho vstupní a výstupní rozhraní. Poté byl popsán jeden z možných způsobů propojení procesoru s obrazovým senzorem. Dále jsem se zabýval implementací základních algoritmů pro zpracování obrazu a jejich optimalizací pro procesor Blackfin. Na závěr jsem porovnal výkonnost algoritmů před a po optimalizaci, a také provedl srovnání výkonnosti procesoru Blackfin s běžným procesorem pro osobní počítače.

Zadání bakalářské práce tak bylo splněno, ale toto téma ještě určitě není úplně vyčerpané. Cílem práce nebylo vytvořit kompletní funkční systém. Další vývoj by tak závisel na požadavcích konkrétního nasazení.

Popsané grafické algoritmy představují pouze zlomek možností digitálního zpracování obrazu. Kromě jednoduchých filtrů by bylo zajímavé implementovat i jiné typy algoritmů, například algoritmy sloužící pro rozpoznávání prvků v obraze. Takové systémy se mohou uplatnit například při automatickém sledování dopravní situace, kdy je možné v obrazu nalézt a přečíst poznávací značku automobilu.

Provedená optimalizace grafických algoritmů přinesla znatelné zlepšení výkonu. Na druhou stranu předpokládám, že zde ještě určitý prostor pro další zrychlení je. Pokud by se ukázalo, že ani provedené úpravy nejsou dostatečné, je zde stále možnost přepsat některé části kódu do assembleru.

Tato práce by případnému zájemci měla poskytnout ucelený pohled na vývoj systémů pro digitální zpracování obrazu, postavených na platformě procesoru Blackfin. Firma Analog Devices, která tyto procesory vyrábí, poskytuje na svých webových stránkách velmi podrobnou dokumentaci ke svým produktům. Ta je však v anglickém jazyce. Taktéž k problematice zpracování obrazových signálů lze najít velké množství informací, jak v anglickém, tak i v českém jazyce. Velká část z nich je však napsána poměrně složitě a předpokládá pokročilé znalosti matematiky.

Doufám, že tato práce poslouží případným zájemců o problematiku digitálního zpracování obrazu jako vstupní bod a prvotní zdroj informací.

10. Bibliografie

1. **Analog Devices.** *Blackfin Processor Programming Reference.* 2008.

2. —. *Blackfin ADSP-BF534/ADSP-BF536/ADSP-BF537 Datasheet.* 2010.

3. —. *ADSP-BF537 EZ-KIT Lite Evaluation System Manual.* 2008.

4. —. *VisualDSP++ 5.0 Getting Started Guide.* 2007.

5. **Kodak.** *Kodak KAC-9618 CMOS Image Sensor Device Performance Specification.* 2007.

6. **Michal, Dobeš.** *Zpracování obrazu a algoritmy v C#.* místo neznámé : BEN, 2008. 978-80-7300-2.

7. **Analog Devices.** *Cycle Counting and Profiling.* 2008.

8. —. *Tuning C Source Code for the Blackfin Processor Compiler.* 2003.

9. —. *C/C++ Compiler and Library Manual for Blackfin Processors.* 2010.

10. **CPU World.** CPU World. [Online] 2011. [Citace: 21. Duben 2011.] http://www.cpu-world.com/CPUs/Celeron_Dual-Core/Intel-Mobile%20Celeron%20Dual-Core%20T1400%20LF80537NE030512.html.